AD-A214 955

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Ada Compiler Validation Summary Report: Rational, MC68020 Family Cross Development Facility, Version 5, R1000 Series 200 Model 20 (Host) to Motorola 68020 in MVME 135 board (Target), 890712W1.10112		5. TYPE OF REPORT & PERIOD COVERED 12 July 1989 to 12 July 1990 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Wright-Patterson AFB Dayton, OH, USA		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS Wright-Patterson AFB Dayton, OH, USA		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081		12. REPORT DATE 13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office) Wright-Patterson AFB Dayton, OH, USA		15. SECURITY CLASS (of this report) UNCLASSIFIED 15a. DECLASSIFICATION DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20. If different from Report)

UNCLASSIFIED

DTIC ELECTE DEC 0 4 1989 S B D

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Rational, MC68020 Family Cross Development Facility, Version 5, Wright-Patterson AFB, R1000 Series 200 Model 20 under Rational Environment, Version D_11_0_8 (Host) to Motorola 680 in MVME 135 board (bare machine)(Target), ACVC 1.10.

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73 S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

89 11 30 054

Ada Compiler Validation Summary Report:

Compiler Name: MC68020 Family Cross Development Facility, Version 5

Certificate Number: 890712W1.10112

Host: R1000 Series 200 Model 20 under
Rational Environment, Version D_11_0_8

Target: Motorola 68020 in MVME 135 board
(bare machine)

Testing Completed 12 July 1989 Using ACVC 1.10

This report has been reviewed and is approved.

Steve P. Wilson

Ada Validation Facility
Steve P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

J F Kramer

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311

John P Solomond

Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington DC 20301

AVF Control Number: AVF-VSR-292.0789
89-04-13-RAT

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 890712W1.10112
Rational
MC68020 Family Cross Development Facility, Version 5
R1000 Series 200 Model 20 Host and Motorola 68020 in MVME 135 board Target

Completion of On-Site Testing:
12 July 1989

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation-dependent but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard

- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard

- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc. under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 12 July 1989 at Santa Clara CA.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C.#552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including

cross-compilers, translators, and interpreters.

Failed test — An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.

Host — The computer on which the compiler resides.

Inapplicable test — An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.

Passed test — An ACVC test for which a compiler generates the expected result.

Target — The computer for which a compiler generates code.

Test — A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.

Withdrawn test — An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation of legal Ada programs with certain language constructs which cannot be verified at compile time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every

illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate

tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: MC68020 Family Cross Development Facility, Version 5

ACVC Version: 1.10

Certificate Number: 890712W1.10112

Host Computer:

Machine:	R1000 Series 200 Model 20
Operating System:	Rational Environment Version D_11_0_8
Memory Size:	32 Megabytes

Target Computer:

Machine:	Motorola 68020 in MVME 135 board
Operating System:	bare machine
Memory Size:	1 Megabyte

Communications Network: RS232 serial line

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

a. Capacities.

(1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)

(2) The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)

(3) The compiler correctly processes tests containing block statements nested to 65 levels. (See test D56001B.)

(4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 10 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

(1) This implementation supports the additional predefined types SHORT_INTEGER, LONG_FLOAT, and SHORT_SHORT_INTEGER in package STANDARD. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

(1) None of the default initialization expressions for record components are evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)

(2) Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

(3) This implementation uses no extra bits for extra precision and uses all extra bits for extra range. (See test C35903A.)

(4) Sometimes NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

(5) NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

(6) Underflow is not gradual. (See tests C45524A..Z.)

d. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

(1) The method used for rounding to integer is round to even. (See tests C46012A..Z.)

(2) The method used for rounding to longest integer is round to even. (See tests C46012A..Z.)

(3) The method used for rounding to integer in static universal real expressions is round away from zero. (See test C4A014A.)

e. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR f an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT.

For this implementation:

(1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises NUMERIC_ERROR. (See test C36003A.)

(2) NUMERIC_ERROR is raised when a null array type with INTEGER'LAST + 2 components is declared. (See test C36202A.)

(3) NUMERIC_ERROR is raised when a null array type with SYSTEM.MAX_INT + 2 components is declared. (See test C36202B.)

(4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises NUMERIC_ERROR when the array type is declared. (See test C52103X.)

(5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array type is declared. (See test C52104Y.)

(6) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared. (See test E52103Y.)

(7) In assigning one-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

(8) In assigning two-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

f. Discriminated types.

(1) In assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. Aggregates.

(1) In the evaluation of a multi-dimensional aggregate, all choices are evaluated before checking against the index type. (See tests C43207A and C43207B.)

(2) In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

(3) CONSTRAINT_ERROR is raised before all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

h. Pragmas.

(1) The pragma INLINE is supported for functions and procedures. (See tests LA3004A..B, EA3004C..D, and CA3004E..F.)

i. Generics

(1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)

(2) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

j. Input and output

(1) The package SEQUENTIAL_IO cannot be instantiated with unconstrained array types or record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

(2) The package DIRECT_IO cannot be instantiated with unconstrained array types or record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

(3) Sequential, Direct, and Text files are not supported by this implementation.

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 583 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation and 238 executable tests that use file operations not supported by the implementation . Modifications to the code, processing, or grading for 78 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	127	1132	1754	15	16	46	3090
Inapplicable	2	6	561	2	12	0	583
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER 2	3	4	5	6	7	8	9	10	11	12	13	14	TOTAL
Passed	198	577	543	245	171	99	162	331	137	36	252	265	74	3090
Inappl	14	72	137	3	1	0	4	1	0	0	0	104	247	583
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 44 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

E28005C	A39005G	B97102E	C97116A	BC3009B	CD2A62D
CD2A63A	CD2A63B	CD2A63C	CD2A63D	CD2A66A	CD2A66B
CD2A66C	CD2A66D	CD2A73A	CD2A73B	CD2A73C	CD2A73D
CD2A76A	CD2A76B	CD2A76C	CD2A76D	CD2A81G	CD2A83G
CD2A84M	CD2A84N	CD2B15C	CD2D11B	CD5007B	CD50110
ED7004B	ED7005C	ED7005D	ED7006C	ED7006D	CD7105A
CD7203B	CD7204B	CD7205C	CD7205D	CE2107I	CE3111C
CE3301A	CE3411B				

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 583 tests were inapplicable for the reasons indicated:

a. The following 201 tests are not applicable because they have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y	C35705L..Y	C35706L..Y	C35707L..Y
C35708L..Y	C35802L..Z	C45241L..Y	C45321L..Y
C45421L..Y	C45521L..Z	C45524L..Z	C45621L..Z

C45641L..Y   C46012L..Z

b. C35702A and B86001T are not applicable because this implementation supports no predefined type SHORT_FLOAT.

c. The following 16 tests are not applicable because this implementation does not support a predefined type LONG_INTEGER:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45614C	C45631C
C45632C	B52004D	C55B07A	B55B09C	B86001W
CD7101F				

d. C45531M..P (4 tests) and C45532M..P (4 tests) are not applicable because the value of SYSTEM.MAX_MANTISSA is less than 47.

e. C4A013B is not applicable because the evaluation of an expression involving 'MACHINE_RADIX applied to the most precise floating-point type would raise an exception; since the expression must be static, it is rejected at compile time.

f. D4A004B is not applicable because this implementation does not support a static universal expression with a value that lies outside of the range SYSTEM.MIN_INT ... SYSTEM.MAX_INT.

g. D64005G is not applicable because this implementation does not support nesting 17 levels of recursive procedure calls.

h. B86001Y is not applicable because this implementation supports no predefined fixed-point type other than DURATION.

i. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

j. C96005B is not applicable because there are no values of type DURATION'BASE that are outside the range of DURATION.

k. CD1009C, CD2A41A..B (2 tests), CD2A41E, and CD2A42A..J (10 tests) are not applicable because this implementation does not support size clauses for floating point types unless the size given is the same as would have been chosen by the compiler.

l. CD2A61I and CD2A61J are not applicable because this implementation does not support size clauses for array types, which imply compression, with component types of composite or floating point types. This implementation requires an explicit size clause on the component type.

m. CD2A84B..I (8 tests) and CD2A84K..L (2 tests) are not applicable because this implementation does not support size clauses for access types unless the size given is 32.

n. CD2B15B is not applicable because this implementation allocates more memory to collection size than is asked for by the test.

o. The following 76 tests are not applicable because, for this implementation, address clauses may be used only in static scopes:

CD5003B..I (8)	CD5011A..I (9)	CD5011K..N (4)
CD5011Q..S (3)	CD5012A..J (10)	CD5012L..M (2)
CD5013A..I (9)	CD5013K..O (5)	CD5013R..S (2)
CD5014A..O (15)	CD5014R..Z (9)	

p. AE2101C, EE2201D, and EE2201E use instantiations of package SEQUENTIAL_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

q. AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

r. The following 238 tests are inapplicable because sequential, text, and direct access files are not supported:

CE2102A..C(3)	CE2102G..H(2)	CE2102K
CE2102N..Y(12)	CE2103C..D(2)	CE2104A..D(4)
CE2105A..B(2)	CE2106A..B(2)	CE2107A..H(8)
CE2107L	CE2108A..B(2)	CE2108C..H(6)
CE2109A..C(3)	CE2110A..D(4)	CE2111A..I(9)
CE2115A..B(2)	CE2201A..C(3)	CE2201F..N(9)
CE2204A..D(4)	CE2205A	CE2208B
CE2401A..C(3)	CE2401E..F(2)	CE2401H..L(5)
CE2404A..B(2)	CE2405B	CE2406A
CE2407A..B(2)	CE2408A..B(2)	CE2409A..B(2)
CE2410A..B(2)	CE2411A	CE3102A..B(2)
EE3102C	CE3102F..H(3)	CE3102J..K(2)
CE3103A	CE3104A..C(3)	CE3107B
CE3108A..B(2)	CE3109A	CE3110A
CE3111A..B(2)	CE3111D..E(2)	CE3112A..D(4)
CE3114A..B(2)	CE3115A	EE3203A
CE3208A	EE3301B	CE3302A
CE3305A	CE3402A	EE3402B
CE3402C..D(2)	CE3403A..C(3)	CE3403E..F(2)
CE3404B..D(3)	CE3405A	EE3405B
CE3405C..D(2)	CE3406A..D(4)	CE3407A..C(3)
CE3408A..C(3)	CE3409A	CE3409C..E(3)
EE3409F	CE3410A	CE3410C..E(3)
EE3410F	CE3411A	CE3411C
CE3412A	EE3412C	CE3413A
CE3413C	CE3602A..D(4)	CE3603A
CE3604A..B(2)	CE3605A..E(5)	CE3606A..B(2)
CE3704A..F(6)	CE3704M..O(3)	CE3706D
CE3706F..G(2)	CE3804A..P(16)	CE3805A..B(2)

CE3806A..B(2)	CE3806D..E(2)	CE3806G..H(2)
CE3905A..C(3)	CE3905L	CE3906A..C(3)
CE3906E..F(2)		

s. CE2103A, CE2103B, and CE3107A are not applicable because this implementation does not support external file CREATE and OPEN operations. These tests terminate with an unhandled exception. (See Section 3.6.)

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 78 tests.

The following tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B22003A	B22003B	B22004A	B22004B	B22004C	B23004A
B23004B	B24001A	B24001B	B24001C	B24005A	B24005B
B24007A	B24009A	B24204B	B24204C	B24204D	B25002B
B26001A	B26002A	B26005A	B28003A	B28003C	B29001A
B2A003B	B2A003C	B2A003D	B2A007A	B32103A	B33201B
B33202B	B33203B	B33301B	B35101A	B36002A	B36201A
B37205A	B37307B	B38003A	B38003B	B38009A	B38009B
B41201A	B41202A	B44001A	B44004B	B44004C	B45205A
B48002A	B48002D	B51001A	B51003A	B51003B	B53003A
B55A01A	B64001B	B64006A	B67001H	B74003A	B91001H
B95001C	B95003A	B95004A	B95079A	BB3005A	BC1303F
BC2001D	BC2001E	BC3003A	BC3003B	BC3005B	BC3013A
BD5008A					

C45651A required evaluation modification because the test contains an if statement with a range that excludes some allowable values and FAILED may be called. The AVO has ruled that the failure message "ABS 928.0 NOT IN CORRECT RANGE" may be ignored and the test graded as passed.

D4A004B was rejected at compile time because it contains a static universal expression with a value that lies outside of the range SYSTEM.MIN_INT ... SYSTEM.MAX_INT. The AVO has ruled this test as not applicable to this implementation.

CE2103A, CE2103B, and CE3107A required evaluation modification because these tests do not allow the implementation to raise USE_ERROR for external file CREATE and OPEN operations when these operations are not supported by the implementation. Execution of these tests terminates with an unhandled exception. The AVO has ruled these tests as not applicable to this implementation.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the MC68020 Family Cross Development Facility was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the MC68020 Family Cross Development Facility using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host computer:	R1000 Series 200 Model 20
Host operating system:	Rational Environment, Version D_11_0_8
Target computer:	Motorola 68020 in MVME 135 board
Target operating system:	bare machine
Compiler:	MC68020 Family Cross Development Facility, Version 5

The host and target computers were linked via RS232 serial line.

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded to disk, the full set of tests was compiled and linked on the R1000 Series 200 Model 20, then all executable images were transferred to the Motorola 68020 in MVME 135 board via RS232 serial line and run. Results were printed from the host computer.

The compiler was tested using command scripts provided by Rational and reviewed by the validation team. The compiler was tested using all default option settin except for the following:

OPTION	EFFECT
Create_Subprogram_Specs := False	Missing subprogram specs are not automatically created when a subprogram body is added to the program library.

Tests were compiled, linked, and executed (as appropriate) using a single host and a single target computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at Santa Clara CA and was completed on 12 July 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

Rational has submitted the following Declaration of Conformance concerning the MC68020 Family Cross Development Facility.

DECLARATION OF CONFORMANCE

Compiler Implementor: Rational
Ada Validation Facility: ASD/SCEL, Wright-Patterson AFB OH 45433-6503
Ada Compiler Validation Capability (ACVC) Version: 1.1

Base Configuration

Base Compiler Name: M68020 Family Cross Development Facility Version: 5
Host Architecture: R1000 Series 200, Model 20
Operating System: Rational Environment Version D_11_0_8

Target Architecture: Motorola 68020 in MVME 135 board
Operating System: Bare Machine

Implementor's Declaration

I, the undersigned, representing Rational, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler listed in this declaration. I declare that Rational is the owner of record of the Ada language compilers listed above and, as such, is responsible for maintaining said compiler in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compilers(s) listed in this declaration shall be made only in the owner's corporate name.

Date: 7/10/85

Rational
David H. Bernstein
Vice President, Product Development

Owners Declaration

I, the undersigned, representing Rational, take full responsibility for implementation and maintenance of the Ada compiler(s) listed above, and agree to the public disclosure of the final Validation Summary Report. I declare that all of the Ada language compilers listed, and their host/target performance are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A

Date: 7/10/85

Rational
David H. Bernstein
Vice President, Product Development

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the MC68020 Family Cross Development Facility, Version 5, as described in this Appendix, are provided by Rational. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

   ...

   type INTEGER is range -2147483648 .. 2147483647;
   type SHORT_INTEGER is range -32768 .. 32767;
   type SHORT_SHORT_INTEGER is range -128 .. 127;

   type FLOAT is digits 6 range
     -16#1.FFFFFE# * 2.0 ** 127 .. 16#1.FFFFFE# * 2.0 ** 127;
   type LONG_FLOAT is digits 15 range
     -16#1.FFFFFFFFFFFFF# * 2.0 ** 1023 .. 16#1.FFFFFFFFFFFFF# * 2.0 ** 1023;

   type DURATION is delta 16#1.0# * 2.0 ** (-14) range
     -16#1.0# * 2.0 ** 17 .. 16#1.FFFFFFFC# * 2.0 ** 16;

   ...

end STANDARD;
```

Appendix F to the LRM for the Mc68020_Bare Target

The *Reference Manual for the Ada Programming Language* (LRM) specifies that certain features of the language are implementation-dependent. It requires that these implementation dependencies be defined in an appendix called Appendix F. This is Appendix F for the Mc68020_Bare target, compiler Version 5. It contains materials on the following topics listed for inclusion by the LRM on page F-1:

- Implementation-dependent pragmas
- Implementation-dependent attributes
- Package System
- Representation clauses
- Implementation-dependent components
- Interpretation of expressions that appear in address clauses
- Unchecked conversion
- Implementation-dependent characteristics of I/O Packages

These topics appear in section and subsection titles of this appendix. The appendix contains other topics mentioned in the LRM as being implementation dependent. For these, a reference to the LRM is given in the section or subsection title.

IMPLEMENTATION-DEPENDENT PRAGMAS

The M68020 Family cross-compiler supports pragmas for application software development in addition to those listed in Annex B of the LRM. They are described below, along with additional clarifications and restrictions for pragmas defined in Annex B of the LRM.

Pragma Main

A parameterless library-unit procedure without subunits can be designated as a main program by including a pragma Main at the end of the unit specification or body. This pragma causes the linker to run and create an executable program when the body of this subprogram is coded. Before a unit having a pragma Main can be coded, all units in the *with* closure of the unit must be coded.

The pragma Main has three arguments:

- **Target**. A string specifying the target key. If this argument appears and it does not match the current target key, the pragma Main is ignored If the Target parameter matches the current target key or does not appear, pragma Main is honored. A single source copy of a main program may be used for different targets by putting in multiple Main pragmas with different target parameters and different stack_sizes and/or different heap_sizes.
- **Stack_Size**: A static integer expression specifying the size in bytes of the main task stack. If not specified, the default value is 4K bytes.
- **Heap_Size**: A static integer expression specifying the size in bytes of the heap. If not specified, the default value is 64K bytes.

The complete syntax for this pragma is:

```
pragma_main   ::=  PRAGMA MAIN
                   [ ( main_option { , main_option } ) ] ;

main_option   ::=  TARGET => simple_name |
                   STACK_SIZE => static_integer_expression |
                   HEAP_SIZE => static_integer_expression
```

The pragma Main must appear immediately after the declaration or body of a parameterless library-unit procedure without subunits.

Pragma Nickname

The pragma Nickname can be used to give a unique string name to a procedure or function in addition to its normal Ada name. This unique name can be used to distinguish among overloaded procedures or functions in the importing and exporting pragmas defined in subsequent sections.

The pragma Nickname must appear immediately following the declaration for which it is to provide a nickname. It has a single argument, the nickname, which must be a string constant

For example:

```
function Cat (L: Integer; R: String) return String;
pragma Nickname ("Int-Str-Cat");

function Cat (L: String; R: Integer) return String;
pragma Nickname ("Str-Int-Cat");

pragma Interface (Assembly, Cat);

pragma Import_Function (Internal => Cat,
                        Nickname => "Int-Str-Cat",
                        External => "CAT$INT_STR_CONCAT",
                        Mechanism => (Value, Reference));

pragma Import_Function (Internal => Cat,
                        Nickname => "Str-Int-Cat",
                        External => "CAT$STR_INT_CONCAT",
```

```
Mechanism => (Reference, Value));
```

Pragmas Import_Procedure and Import_Function

A subprogram written in another language (typically, assembly language) can be called from an Ada program if it is declared with a pragma Interface. The rules for placement of pragma Interface are given in Section 13.9 of the LRM. Every interfaced subprogram must have an importing pragma recognized by the M68020 Family cross-compiler, either Import_Procedure or Import_Function. These pragmas are used to declare the external name of the subprogram and the parameter-passing mechanism for the subprogram call. If an interfaced subprogram does not have an importing pragma, or if the importing pragma is incorrect, pragma interface is ignored.

Importing pragmas can be applied only to nongeneric procedures and functions.

The pragmas Import_Procedure and Import_Function are used for importing subprograms. Import_Procedure is used to call a non-Ada procedure; Import_Function, a non-Ada function.

Each import pragma must be preceded by a pragma Interface; otherwise, the placement rules for these pragmas are identical to those of the pragma Interface.

The importing pragmas have the form:

```
importing_pragma   ::=  PRAGMA importing_type
                        ( [ INTERNAL => ] internal_name
                          [ , [ EXTERNAL => ] external_name ]
                          [ [ , [ PARAMETER_TYPES => ]
                                                  parameter_types ]
                            [ , [ RESULT_TYPE => ] type_mark ] ]
                            [ , NICKNAME => string_literal ]
                          [ , [ MECHANISM => ] mechanisms ] ) ;

importing_type     ::=  IMPORT_PROCEDURE | IMPORT_FUNCTION |
                        IMPORT_VALUED_PROCEDURE
internal_name      ::=  identifier |
                        string_literal   -- An operator designator

external_name      ::=  identifier | string_literal

parameter_types    ::=  ( NULL ) | ( type_mark { , type_mark } )

mechanisms         ::=  mechanism_name |
                        ( mechanism_name { , mechanism_name } )

mechanism_name     ::=  VALUE | REFERENCE
```

The internal name is the Ada name of the subprogram being interfaced. If more than one subprogram is in the declarative region preceding the importing pragma, the correct subprogram must be identified by either using the argument types (and result type, if a function) or specifying the nickname.

If it is used to identify a subprogram with an overloaded internal name, the value of the Parameter_Types argument consists of a list of type or subtype names, not names of parameters. Each one corresponds positionally to a formal parameter in the subprogram's declaration. If the subprogram has no parameters, the list consists of the single word *null*. For a function, the value of the Result_Type argument is the name of the type returned by the function.

The external designator, specified with the External parameter, is a character string that is an identifier suitable for the MC68020 assembler. If the external designator is not specified, the internal name is used.

The Mechanism argument is required if the subprogram has any parameters. The argument specifies, in a parenthesized list, the passing mechanism for each parameter to be passed. There must be a mechanism specified for each parameter listed in **parameter_types** and they must correspond positionally. The types of mechanism are as follows.

- **Value**: Specifies that the parameter is passed on the stack by immediate value.
- **Reference**: Specifies that the parameter is passed on the stack by address. Used for structures having many values.

For functions, it is not possible to specify the passing mechanism of the function result; the standard Ada mechanism for the given type of the function result must be used by the interfaced subprogram. If there are parameters, and they all use the same passing mechanism, then an alternate form for the Mechanism argument can be used: instead of a parenthesized list with an element for each parameter, the single mechanism name (not parenthesized) can be used instead.

Examples:

```
procedure Locate (Source: in String;
                  Target: in String;
                  Index:  out Natural);

pragma Interface (Assembler, Locate);
pragma Import_Procedure
          (Internal => Locate,
           External => "STR$LOCATE",
           Parameter_Types => (String, String, Natural),
           Mechanism => (Reference, Reference, Value));


function Pwr (I: Integer; N: Integer) return Float;
function Pwr (F: Float; N: Integer) return Float;

pragma Interface (Assembler, Pwr);

pragma Import_Function
          (Internal => Pwr,
           Parameter_Types => (Integer, Integer),
           Result_Type => Float,
```

```
               Mechanism => Value,
               External => "MATH$PWR_OF_INTEGER");
pragma Import_Function
              (Internal => Pwr,
               Parameter_Types => (Float, Integer),
               Result_Type => Float,
               Mechanism => Value,
               External => "MATH$PWR_OF_FLOAT");
```

Pragmas Export_Procedure and Export_Function

A subprogram written in Ada can be made accessible to code written in another language by using an exporting pragma defined by the M68020 Family cross-compiler. The effect of such a pragma is to give the subprogram a defined symbolic name that the linker can use when resolving references between object modules.

Exporting pragmas can be applied only to nongeneric procedures and functions.

An exporting pragma can be given only for subprograms that are library units or that are declared in the specification of a library package. An exporting pragma can be placed after a subprogram body only if the subprogram does not have a separate specification. Thus, an exporting pragma cannot be applied to the body of a library subprogram that has a separate specification.

These pragmas have similar arguments to the importing pragmas, except that it is not possible to specify the parameter-passing mechanism. The standard Ada parameter-passing mechanisms are chosen. For descriptions of the pragma's arguments (Internal, External, Parameter_Types, Result_Type, and Nickname), see the preceding section on the importing pragmas.

The full syntax of the pragmas for exporting subprograms is:

```
exporting_pragma  ::=  PRAGMA exporting_type
                       ( [ INTERNAL => ] internal_name
                         [ , [ EXTERNAL => ] external_name ]
                         [ [ , [ PARAMETER_TYPES => ] parameter_types ]
                           [ , [ RESULT_TYPE => ] type_mark ] |
                           [ , NICKNAME => string_literal ] ] ) ;
exporting_type    ::=  EXPORT_PROCEDURE | EXPORT_FUNCTION
internal_name     ::=  identifier |
                       string_literal   -- An operator designator
external_name     ::=  identifier | string_literal
parameter_types   ::=  ( NULL ) | ( type_mark { , type_mark } )
```

Examples:

```
procedure Matrix_Multiply
             (A, B: in Matrix; C: out Matrix);

pragma Export_Procedure (Matrix_Multiply);
-- External name is the string "Matrix_Multiply"
```

```
function Sin (R: Radians) return Float;
pragma Export_Function
            (Internal => Sin,
             External => "SIN_RADIANS");
-- External name is the string "SIN_RADIANS"
```

Pragma Export_Elaboration_Procedure

The pragma Export_Elaboration_Procedure makes the elaboration procedure for a given compilation unit available to external code by defining a global symbolic name. This procedure is otherwise unnamable by the user. Its use is confined to the exceptional circumstances where an Ada module is not elaborated because it is not in the closure of the main program or if the main program is not an Ada program. This pragma is not recommended for use in application programs unless the user has a thorough understanding of elaboration, runtime and storage model considerations.

The pragma Export_Elaboration_Procedure must appear immediately following the compilation unit.

The complete syntax for this pragma is:

```
pragma_export_elaboration_procedure ::=
    PRAGMA EXPORT_ELABORATION_PROCEDURE ( EXTERNAL_NAME => external_name );

external_name ::= identifier | string_literal
```

Pragmas Import_Object and Export_Object

Objects can be imported or exported from an Ada unit with the pragmas Import_Object and Export_Object. The pragma Import_Object causes an Ada name to reference storage declared and allocated in some external (non-Ada) object module. The pragma Export_Object provides an object declared within an Ada unit with an external symbolic name that the linker can use to allow another program to access the object. It is the responsibility of the programmer to ensure that the internal structure of the object and the assumptions made by the importing code and data structures correspond. The cross-compiler cannot check for such correspondence.

The object to be imported or exported must be a variable declared at the outermost level of a library package specification or body.

The size of the object must be static. Thus, the type of the object must be one of:

- A scalar type (or subtype)
- An array subtype with static index constraints whose component size is static
- A simple record type or subtype

Objects of a private or limited private type can be imported or exported only into the package that declares the type.

Imported objects cannot have an initial value and thus cannot be:

- A constant
- An access type
- A task type
- A record type with discriminants, with components having default initial expressions, or with components that are access types or task types

In addition, the object must not be in a generic unit. The external name specified must be suitable as an identifier in the assembler.

The full syntax for the pragmas Import_Object and Export_Object is:

```
object_pragma       ::=  PRAGMA object_pragma_type
                           ( [ INTERNAL => ] identifier
                             [ , [ EXTERNAL => ] string_literal ] ) ;

object_pragma_type  ::=  IMPORT_OBJECT | EXPORT_OBJECT
```

Pragma Suppress_All

This pragma is equivalent to the following sequence of pragmas:

```
pragma Suppress (Access_Check);
pragma Suppress (Discriminant_Check);
pragma Suppress (Division_Check);
pragma Suppress (Elaboration_Check);
pragma Suppress (Index_Check);
pragma Suppress (Length_Check);
pragma Suppress (Overflow_Check);
pragma Suppress (Range_Check);
pragma Suppress (Storage_Check);
```

Note that, like pragma Suppress, pragma Suppress_All does not prevent the raising of certain exceptions. For example, numeric overflow or dividing by zero is detected by the hardware, which results in the predefined exception Numeric_Error. Refer to Chapter 5, "Runtime Organization," for more information.

Pragma Suppress_All must appear immediately within a declarative part.

Pragma Interrupt_Handler and Address Clauses for Interrupt Handling

Three different mechanisms are available to support interrupt handling: address clauses on task entries; subprograms identified with pragma Interrupt_Handler; and interrupt-handling queues that employ both subprograms and task entries.

Simple interrupt handling can be accomplished with address clauses attached to task entries, as described in the LRM (Section 13.5.1). Note that the task entry must always be available.

As one alternative, interrupt-handling subprograms can be called on some nonspecific, target-dependent thread. The subprograms run at interrupt level and with some restrictions.

These handlers may make use of the Runtime_Interface package to control various aspects of certain tasks. Each interrupt-handling procedure must have a single formal parameter of type Standard.Integer. The actual value of this parameter and interpretation of it during execution of the handler is target-dependent.

A pragma associates an interrupt-handling subprogram with a corresponding interrupt vector. More than one pragma can be applied to a single subprogram if the handler is to be associated with more than a single interrupt source. The syntax for the pragma is as follows:

```
procedure Handler_Procedure (Target_Dependent_Parameter : Integer);

pragma Interrupt_Handler (Handler => Handler_Procedure,
                          Vector  => [address-expression]);
```

The vector parameter is interpreted by the runtime system. The cross-compiler requires that the subprogram be declared at the outermost scope. The subprogram will be called directly and no elaboration check will be performed, even if elaboration checks are enabled. The pragma may follow either the specification or the body of the subprogram.

The third alternative for interrupt handling is the use of queued interrupts, a combination of the first two approaches. With queued interrupts, a subprogram is called for some functions that must happen at the time of the interrupt; it must also clear the interrupt request. After the subprogram returns to the runtime system, an entry call is enqueued to an associated task entry so that the entry will be accepted as soon as the task attempts to accept it. Both the subprogram and the entry must have a single formal parameter of type Standard.Integer, whose interpretation during execution of the subprogram or rendezvous is target-dependent.

A pragma associates the interrupt-handling subprogram with the task entry and the interrupt vector. More than one pragma may be associated with a given subprogram and/or task entry if that subprogram and/or task entry are to serve as handlers for more than one interrupt vector. The syntax for the pragma is:

```
type Driver is

  entry Handler (Target_Dependent_Parameter : integer);

end Driver;

T : Driver;

procedure Interrupt_Handler (Target_Dependent_Parameter : Integer);

pragma Interrupt_Handler (Handler => Interrupt_Handler,
                             Vector => [address-expression],
                             Task_Entry => T.Handler);
```

There is no address clause on the entry in the task specification.

The compiler ensures that the associated handler and the named task object T are declared at the outermost scope. The subprogram will be called directly and no elaboration check will be performed, even if elaboration checks are enabled. The task object must be activated prior to

receiving the first interrupt. No check is performed at runtime for this prefix condition.

In the event that a task entry calls an interrupt entry using normal entry-calling mechanisms, that entry call will not be accepted by the called task until the called task accepts the given entry and no interrupts are pending. This happens even when the normal entry call happens prior to the interrupt. In other words, although normal entry calls are serviced with a FIFO queue, all pending interrupt entry calls are processed before *any* normal entry calls.

The associated procedure is called by the runtime system at interrupt level on an interrupt stack. The context at the time of the call is the context at the time of the interrupt, not the context of the associated task. The interrupt-handling procedure is responsible for clearing the interrupt and performing any device-specific actions required. The interrupt-handling procedure must conform to a set of restrictions that have not been fully defined but include not using any tasking features of the language and not raising exceptions. The predefined package Calendar can be used and dynamic memory allocation/deallocation is allowed. No checks are performed to ensure that restrictions are not violated, and such violations may have unpredictable results.

When the interrupt-handling procedure returns, the runtime system enqueues an entry call to the associated task entry in such a way that it will be accepted as soon as the task is prepared to accept it. The interrupts are fully buffered and the task will accept one entry call for each interrupt without respect to the rate at which interrupts are received. For example, if 10 interrupts are received before the task accepts the first, then the 'Count of the associated entry will be 10, and 10 accept statements for that entry will be required to reduce the 'Count to 0. The priority of the task during the rendezvous will be proportional to the priority of the interrupt and higher than Standard.System.Priority'Last.

The elaboration of pragma Interrupt_Handler has the effect of associating either a task entry or a subprogram with an interrupt vector. This may result in the propagation of the exception Standard.Program_Error if the vector already has an associated handler.

IMPLEMENTATION-DEPENDENT ATTRIBUTES

There are no implementation-dependent attributes.

PACKAGE STANDARD (*LRM Annex C*)

Package Standard defines all the predefined identifiers in the language.

```
package Standard is

    type *Universal_Integer* is ...
    type *Universal_Real* is ...
    type *Universal_Fixed* is ...
    type Boolean is (False, True);
```

```
    type Integer is range -2147483648 .. 2147483647;
    type Short_Short_Integer is range -128 .. 127;
    type Short_Integer is range -32768 .. 32767;

    type Float is digits 6 range -16#1.FFFF_FE# * 2.0 ** 127 ..
                                 16#1.FFFF_FE# * 2.0 ** 127;
    type Long_Float is digits 15 range -16#1.FFFF_FFFF_FFFF_F# * 2.0 ** 1023
                                    .. 16#1.FFFF_FFFF_FFFF_F# * 2.0 ** 1023;

    type Duration is delta 16#1.0# * 2.0 ** (-14)
                    range -16#1.0# * 2.0 ** 17 ..
                            16#1.FFFF_FFFC# * 2.0 ** 16;

    subtype Natural is Integer range 0 .. 2147483647;
    subtype Positive is Integer range 1 .. 2147483647;



    type Character is ...

    type String is array (Positive range <>) of Character;
    pragma Pack (String);

    package Ascii is ...

    Constraint_Error : exception;
    Numeric_Error : exception;
    Storage_Error : exception;
    Tasking_Error : exception;
    Program_Error : exception;

end Standard;
```

The following table shows the default integer and floating-point types:

Supported Integer and Floating-Point Types

Ada Type Name	Size
Short_Short_Integer	8 bits
Short_Integer	16 bits
Integer	32 bits
Float	32 bits
Long_Float	64 bits

Fixed-point types are implemented using the smallest discrete type possible; it may be 8, 16, or 32 bits. Standard.Duration is 32 bits.

PACKAGE SYSTEM (*LRM 13.7*)

```
package System is

     type Address is private;

     type Name is (Mc68020_Bare);

     System_Name : constant Name := Mc68020_Bare;



     Storage_Unit : constant := 8;
     Memory_Size : constant := +(2 ** 31) - 1;

     Min_Int : constant := -(2 ** 31);
     Max_Int : constant := +(2 ** 31) - 1;

     Max_Digits : constant := 15;
     Max_Mantissa : constant := 31;
     Fine_Delta : constant := 1.0 / (2.0 ** 31);
     Tick : constant := 1.0 / 125_000.0;

     subtype Priority is Integer range 0 .. 255;

     function To_Address (Value : Integer) return Address;
     function To_Integer (Value : Address) return Integer;



     function "+" (Left : Address; Right : Integer) return Address;
     function "+" (Left : Integer; Right : Address) return Address;
     function "-" (Left : Address; Right : Address) return Integer;
     function "-" (Left : Address; Right : Integer) return Address;

     function "<" (Left, Right : Address) return Boolean;
     function "<=" (Left, Right : Address) return Boolean;
     function ">" (Left, Right : Address) return Boolean;
     function ">=" (Left, Right : Address) return Boolean;
     --
     --  The functions above are unsigned in nature.  Neither Numeric_Error
     --  nor Constraint_Error will ever be propagated by these functions.
     --
     --  Note that this implies:
     --
     --          To_Address (Integer'First) > To_Address (Integer'Last)
     --
     --  and that:
     --
     --          To_Address (0) < To_Address (-1)
```

```
        --
        --  Also, the unsigned range of Address includes values which are
        --  larger than those implied by Memory_Size.
        --

        Address_Zero : constant Address;

    private

        . . .

    end System;
```

REPRESENTATION CLAUSES AND CHANGES OF REPRESENTATON

The MC68020 CDF support for representation clauses is described in this section with reference to the relevant section of the LRM. Usage of a clause that is unsupported as specified in this section or usage contrary to LRM specification will cause a semantic error unless specifically noted.

Length Clauses (*LRM* 13.2)

Length clauses are supported for the M68020 Family CDF as follows:

- The value in a 'Size clause must be a positive static integer expression. 'Size clauses are supported for all scalar and composite types, including derived types, with the following restrictions:
  - — For all types the value of the size attribute must be greater than equal to the minimum size necessary to store the largest possible value of the type.
  - — For discrete types, the value of the size attribute must be less than or equal to 32.
  - — For fixed types, the value of the size attribute must be less than or equal to 32.
  - — For float types, the size clause can only specify the size the type would have if there were no clause.
  - — For access and task types, the value of the size attribute must be 32.
  - — For composite types, a size specification must not imply compression of composite components. Such compression must have been explicitly requested using a length clause or pragma Pack on the component type.
- 'Storage_Size clauses are supported for access and task types. The value given in a Storage_Size clause may be any integer expression, and it is not required to be static.
- 'Small clauses are supported for fixed point types. The value given in a 'Small clause must be a non-zero static real number.

Enumeration Representation Clauses (*LRM 13.3*)

Enumeration representation clauses are supported with the following restrictions.

- The values given in the clause must be in ascending order.
- Every enumeration literal must have a unique integer value assigned to it.
- The allowable values for an enumeration clause range from (Integer'First + 1) to Integer'Last.
- Negative numbers are allowed.

Record Representation Clauses (*LRM 13.4*)

Both full and partial representation clauses are supported for both discriminated and undiscriminated records. The `static_simple_expression` in the alignment clause part of a record representation clause (see LRM 13.4 (4)) must be a power of two with the following limits: `1 <= static_simple_expression` <= 16 .

The size specified for a discrete field in a component clause must not exceed 32 .

Implementation-Dependent Components

The LRM allows for the generation of names denoting implementation-dependent components in records. For the M68020 Family CDF, there are no such names visible to the user.

Address Clauses (*LRM 13.5*)

Address clauses can be applied to objects. The address must be determinable at compile time, but it need not be Ada static (as defined in the LRM, Section 4.9). The 'Address attribute does not produce a compile-time static value. Addresses must be specified using constants and operators from package System. Objects to which address clauses apply must not appear within anyh subprograms or task body.

Address clauses for interrupts: Address clauses may be attached to a task entry. The task entry *must* be available at the time of the interrupt for this kind of interrupt handling.

Change of Representation (*LRM 13.6*)

Change of representation is supported wherever it is implied by support for representation specifications. In particular, type conversions between array types or record types may cause packing or unpacking to occur; conversions between related enumeration types with different representations may result in table lookup operations.

OTHER IMPLEMENTATION-DEPENDENT FEATURES

Machine Code (*LRM 13.8*)

Machine-code insertions are not supported at this time.

Unchecked Storage Deallocation (*LRM 13.10.1*)

Unchecked storage deallocation is implemented by the generic function `Unchecked_Deallocation` defined by the LRM. This procedure can be instantiated with an object type and its access type resulting in a procdure that deallocates the object's storage. Objects of any type may be deallocated.

The storage reserved for the entire collection is reclaimed when the program exits the scope in which the access type is declared. Placing an access type declaration within a block can be a useful implementation strategy when conservation of memory is necessary. Space on the free list is coalesced when objects are deallocated.

Erroneous use of dangling references may be detected in certain cases. When detected, the exception Storage_Error is raised. Deallocation of objects that were not created through allocation (ie through Unchecked_Conversion) may also be detected in certain cases and raises Storage_Error.

Unchecked Type Conversion (*LRM 13.10.2*)

Unchecked conversion is implemented by the generic function `Unchecked_Conversion` defined by the LRM. This function can be instantiated with *Source* and *Target* types resulting in a function that converts source data values into target data values.

Unchecked conversion moves storage units from the source object to the target object sequentially, starting with the lowest address. Transfer continues until the source object is exhausted or the target object runs out of room. If the target is larger than the source then the remaining bits are undefined. Depending on the target computer architecture, the result of conversions may be right or left aligned.

Restrictions on Unchecked Type Conversion

- The target type of an unchecked conversion cannot be an unconstrained array type or an unconstrained discriminated type without default discriminants.
- Internal consistency among components of the target type is not guaranteed. Discriminant components may contain illegal values or be inconsistent with the use of those discriminants elsewhere in the type representation.

CHARACTERISTICS OF I/O PACKAGES

External files are not supported for Direct_Io, Sequential_Io, or Text_Io. Direct_Io and Sequential_Io may not be instantiated with unconstrained array types or unconstrained record types not having default discriminants.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
$BIG_ID1 An identifier the size of the maximum input line length which is identical to $BIG_ID2 except for the last character.	(1..253 => 'A', 254 => '1')
$BIG_ID2 An identifier the size of the maximum input line length which is identical to $BIG_ID1 except for the last character.	(1..253 => 'A', 254 => '2')
$BIG_ID3 An identifier the size of the maximum input line length which is identical to $BIG_ID4 except for a character near the middle.	(1..126 => 'A', 127 => '3', 128..254 => 'A')

TEST PARAMETERS

Name and Meaning	Value
$BIG_ID4 An identifier the size of the maximum input line length which is identical to $BIG_ID3 except for a character near the middle.	(1..126 => 'A', 127 => '4', 128..254 => 'A')
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..251 => '0', 252..254 => "298")
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..249 => '0', 250..254 => "690.0")
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	(1 => '"', 2..128 => 'A', 129 => '"')
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	(1 => '"', 2..127 => 'A', 128 => '1', 129 => '"')
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(1..234 => ' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	1000000000
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	2147483647
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8

Name and Meaning	Value
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	MC68020_BARE
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	0.000000000465661287307739257812 5
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	2147483647
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_FIXED_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_FLOAT_NAME
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	1.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	131073.0
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	255
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	\NODIRECTORY\FILENAME
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	THIS-FILE-NAME-IS-TOO-LONG-FOR-MY-SYSTEM
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648

TEST PARAMETERS

Name and Meaning	Value
\$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
\$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
\$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-1.0
\$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-131073.0
\$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	0
\$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
\$MAX_DIGITS Maximum digits supported for floating-point types.	15
\$MAX_IN_LEN Maximum input line length permitted by the implementation.	254
\$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
\$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648
\$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..2 => "2:", 3..251 => '0', 252..254 => "11:")

Name and Meaning	Value
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..3 => "16:", 4..250 => '0', 251..254 => "F.E:")
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	(1 => '"', 2..253 => 'A', 254 => '"')
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	32
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	SHORT_SHORT_INTEGER
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	MC68020_BARE
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFE#
$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma MEMORY_SIZE, other than $DEFAULT_MEM_SIZE. If there is no other value, then use $DEFAULT_MEM_SIZE.	2147483647

Name and Meaning	Value
$NEW_STOR_UNIT An integer literal whose value is a permitted argument for pragma STORAGE_UNIT, other than $DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.	8
$NEW_SYS_NAME A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.	MC68020_BARE
$TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has a single entry with one 'IN OUT' parameter.	32
$TICK A real literal whose value is SYSTEM.TICK.	8.0E-06

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 44 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

a. E28005C: This test expects that the string "-- TOP OF PAGE. --63" of line 204 will appear at the top of the listing page due to a pragma PAGE in line 203; but line 203 contains text that follows the pragma, and it is this text that must appear at the top of the page.

b. A39005G: This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).

c. B97102E: This test contains an unintended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).

d. C97116A: This test contains race conditions, and it assumes that guards are evaluated indivisibly. A conforming implementation may use interleaved execution in such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING_OF_THE_GUARD results in a call to REPORT.FAILED at one of lines 52 or 56.

e. BC3009B: This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).

f. CD2A62D: This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).

g. CD2A63A..D, CD2A66A..D, CD2A73A..D, and CD2A76A..D (16 tests): These

tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

h. CD2A81G, CD2A83G, CD2A84M..N, and CD50110 (5 tests): These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86, 96, and 58, respectively).

i. CD2B15C and CD7205C: These tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.

j. CD2D11B: This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.

k. CD5007B: This test wrongly expects an implicitly declared subprogram to be at the address that is specified for an unrelated subprogram (line 303).

l. ED7004B, ED7005C..D, and ED7006C..D (5 tests): These tests check various aspects of the use of the three SYSTEM pragmas; the AVO withdraws these tests as being inappropriate for validation.

m. CD7105A: This test requires that successive calls to CALENDAR.CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK--particular instances of change may be less (line 29).

n. CD7203B and CD7204B: These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

o. CD7205D: This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

p. CE2107I: This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid (line 90).

q. CE3111C: This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

r. CE3301A: This test contains several calls to END_OF_LINE and END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118, 132, and 136).

s. CE3411B: This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.